CSC600 – MODULE 3: LECTURE 8

CONCURRENT PROGRAMMING IN JAVA THREADS I

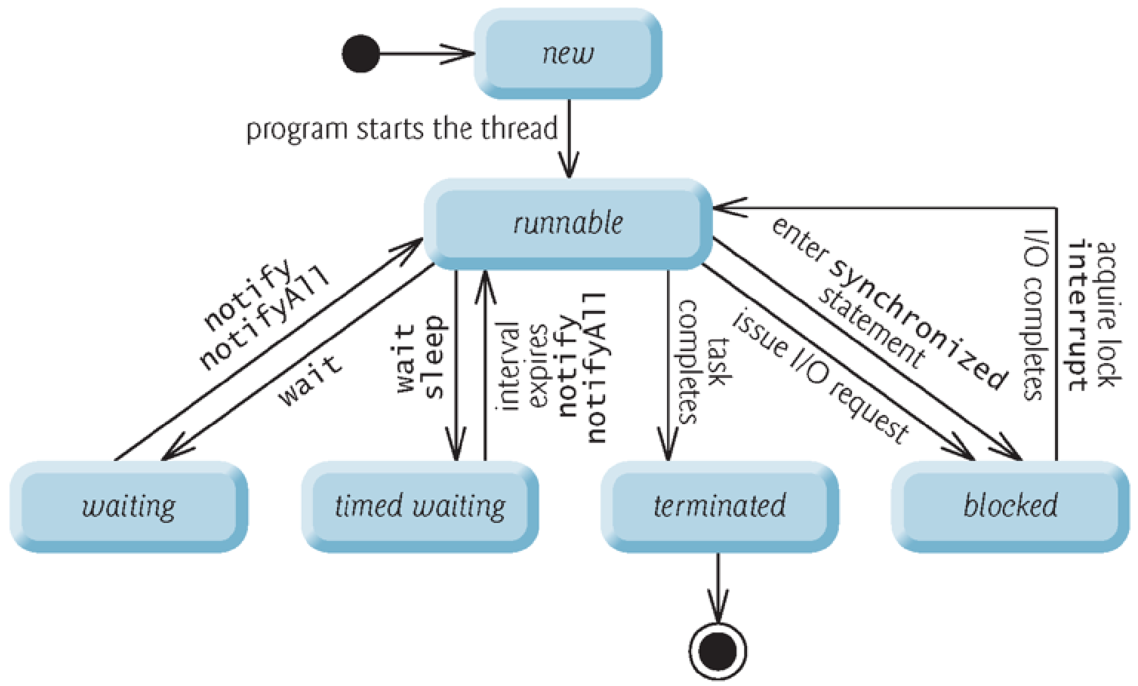
SELECTED EXCERPTS FROM CH. 23
JAVA HOW TO PROGRAM, 11/E

23.1 INTRODUCTION

- When we say that two tasks are operating concurrently, we mean that they’re both *making progress* at once.
- When we say that two tasks are operating in parallel, we mean that they’re executing *simultaneously*.
- Java makes concurrency available to you through the language and APIs.
- You specify that an application contains separate threads of execution
  - each thread has its own method-call stack and program counter
  - can execute concurrently with other threads while sharing application-wide resources such as memory and file handles.
- This capability is called multithreading.

23.2 THREAD STATES AND LIFE CYCLE

- At any time, a thread is said to be in one of several thread states—illustrated in the UML state diagram in Fig. below.
- Let’s look at these in more detail.



23.3 CREATING AND EXECUTING THREADS WITH THE `Executor` FRAMEWORK

- A **`Runnable`** object represents a “task” that can execute concurrently with other tasks.
- The **`Runnable`** interface declares the single method **`run`**, which contains the code that defines the task that a **`Runnable`** object should perform.
- When a thread executing a **`Runnable`** is created and started, the thread calls the **`Runnable`** object’s **`run`** method, which executes in the new thread.

23.3 CREATING AND EXECUTING THREADS WITH THE `Executor` FRAMEWORK

- Class **`PrintTask`** (Fig. 23.3) implements **`Runnable`**, so that multiple **`PrintTasks`** can execute concurrently.
- **`Thread static`** method **`sleep`** places a thread in the *timed waiting state for the specified amount of time.*
    - Can throw a checked exception of type **`InterruptedException`** if the sleeping thread's **`interrupt`** method is called.
- The code in **`main`** executes in the main thread, a thread created by the JVM.
- The code in the **`run`** method of **`PrintTask`** executes in the threads created in **`main`**.
- When method **`main`** terminates, the program itself continues running because there are still threads that are alive.
    - The program will not terminate until its last thread completes execution.

23.3 CREATING AND EXECUTING THREADS WITH THE Executor FRAMEWORK

```java
// Fig. 23.3: PrintTask.java
// PrintTask class sleeps for a random time from 0 to 5 seconds
import java.security.SecureRandom;

public class PrintTask implements Runnable {
   private static final SecureRandom generator = new SecureRandom();
   private final int sleepTime; // random sleep time for thread
   private final String taskName;

   // constructor
   public PrintTask(String taskName) {
      this.taskName = taskName;

      // pick random sleep time between 0 and 5 seconds
      sleepTime = generator.nextInt(5000); // milliseconds
   }

```

Fig. 23.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 1 of 2.)

23.3 CREATING AND EXECUTING THREADS WITH THE Executor FRAMEWORK

```
    // method run contains the code that a thread will execute
    @Override
    public void run() {
       try { // put thread to sleep for sleepTime amount of time
          System.out.printf("%s going to sleep for %d milliseconds.%n",
             taskName, sleepTime);
          Thread.sleep(sleepTime); // put thread to sleep
       }
       catch (InterruptedException exception) {
          exception.printStackTrace();
          Thread.currentThread().interrupt(); // re-interrupt the thread
       }

       // print task name
       System.out.printf("%s done sleeping%n", taskName);
    }
 }
```

Fig. 23.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 2 of 2.)

23.3 CREATING AND EXECUTING THREADS WITH THE Executor FRAMEWORK

```java
// Fig. 23.4: TaskExecutor.java
// Using an ExecutorService to execute Runnables.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class TaskExecutor {
   public static void main(String[] args) {
      // create and name each runnable
      PrintTask task1 = new PrintTask("task1");
      PrintTask task2 = new PrintTask("task2");
      PrintTask task3 = new PrintTask("task3");

      System.out.println("Starting Executor");

      // create ExecutorService to manage threads
      ExecutorService executorService = Executors.newCachedThreadPool();

```

Fig. 23.4 | Using an ExecutorService to execute Runnables. (Part 1 of 4.)

23.3 CREATING AND EXECUTING THREADS WITH THE Executor FRAMEWORK

```
         // start the three PrintTasks
         executorService.execute(task1); // start task1
         executorService.execute(task2); // start task2
         executorService.execute(task3); // start task3

         // shut down ExecutorService--it decides when to shut down threads
         executorService.shutdown();

         System.out.printf("Tasks started, main ends.%n%n");
      }
   }
```

Fig. 23.4 | Using an ExecutorService to execute Runnables. (Part 2 of 4.)

23.3 CREATING AND EXECUTING THREADS WITH THE Executor FRAMEWORK

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 3 of 4.)

23.3 CREATING AND EXECUTING THREADS WITH THE Executor FRAMEWORK

```
Starting Executor
task1 going to sleep for 3161 milliseconds.
task3 going to sleep for 532 milliseconds.
task2 going to sleep for 3440 milliseconds.
Tasks started, main ends.

task3 done sleeping
task1 done sleeping
task2 done sleeping
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnables`. (Part 4 of 4.)

23.3 CREATING AND EXECUTING THREADS WITH THE `Executor` FRAMEWORK

- Recommended that you use the **`Executor`** interface to manage the execution of **`Runnable`** objects for you.
    - Typically creates and manages a group of threads called a thread pool to execute **`Runnables`**.
- **`Executors`** can reuse existing threads and can improve performance by optimizing the number of threads.
- **`Executor`** method **`execute`** accepts a **`Runnable`** as an argument.
- An **`Executor`** assigns every **`Runnable`** passed to its **`execute`** method to one of the available threads in the thread pool.
- If there are no available threads, the **`Executor`** creates a new thread or waits for a thread to become available.

23.3 CREATING AND EXECUTING THREADS WITH THE `Executor` FRAMEWORK

- The **`ExecutorService`** **interface** extends **`Executor`** and declares methods for managing the life cycle of an **`Executor`**.
- An object that implements this interface can be created using **`static`** methods declared in class **`Executors`**.
- **`Executors`** method **`newCachedThreadPool`** returns an **`ExecutorService`** that creates new threads as they're needed by the application.
- **`ExecutorService`** method **`shutdown`** notifies the **`ExecutorService`** to stop accepting new tasks, but continues executing tasks that have already been submitted.

23.4 THREAD SYNCHRONIZATION

- When multiple threads share an object and it is *modified* by one or more of them, indeterminate results may occur unless access to the shared object is managed properly.
- The problem can be solved by giving only one thread at a time *exclusive access* to code that accesses the shared object.
  - During that time, other threads desiring to access the object are kept waiting.
  - When the thread with exclusive access finishes accessing the object, one of the waiting threads is allowed to proceed.
- This process, called thread synchronization, coordinates access to shared data by multiple concurrent threads.
  - Ensures that each thread accessing a shared object *excludes* all other threads from doing so simultaneously—this is called mutual exclusion.

23.4.1 IMMUTABLE DATA

- Thread synchronization is necessary *only* for shared mutable data, i.e., data that may *change* during its lifetime.
- With shared immutable data that will *not* change, it's not possible for a thread to see old or incorrect values as a result of another thread's manipulation of that data.
- When you share *immutable data* across threads, declare the corresponding data fields **`final`** to indicate that the values of the variables will *not* change after they're initialized.
- *Labeling object references as* ***`final`*** *indicates that the reference will not change, but it does not guarantee that the referenced object is immutable—this depends entirely on the object's properties.*
- However, it's still good practice to mark references that will not change as **`final`**.

23.4.2 MONITORS

- A common way to perform synchronization is to use Java’s built-in monitors.
  - Every object has a monitor and a monitor lock (or intrinsic lock).
  - Can be held by a maximum of only one thread at any time.
  - A thread must acquire the lock before proceeding with the operation.
  - Other threads attempting to perform an operation that requires the same lock will be *blocked.*
- To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a `synchronized` statement.
  - Said to be guarded by the monitor lock

23.4.2 MONITORS

- The **`synchronized`** statements are declared using the **`synchronized`** keyword:
    - **`synchronized`** (*object*)
      `{`
        statements
      `}` **`// end synchronized statement`**
- where *object* is the object whose monitor lock will be acquired
    - *object* is normally **`this`** if it's the object in which the **`synchronized`** statement appears.
- When a **`synchronized`** statement finishes executing, the object's monitor lock is released.
- Java also allows **`synchronized`** methods.

23.4.3 UNSYNCHRONIZED MUTABLE DATA SHARING

- A **`SimpleArray`** object (Fig. 23.5) will be *shared* across multiple threads.
- Will enable those threads to place **`int`** values into **`array`**.
- Puts the thread that invokes **`add`** to sleep for a random interval from 0 to 499 milliseconds.
    - This is done to make the problems associated with *unsynchronized access to shared mutable data* more obvious.

23.4.3 UNSYNCHRONIZED MUTABLE DATA SHARING

```java
// Fig. 23.5: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads.
import java.security.SecureRandom;
import java.util.Arrays;

public class SimpleArray { // CAUTION: NOT THREAD SAFE!
   private static final SecureRandom generator = new SecureRandom();
   private final int[] array; // the shared integer array
   private int writeIndex = 0; // shared index of next element to write

   // construct a SimpleArray of a given size
   public SimpleArray(int size) {array = new int[size];}

```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 1 of 3.)

23.4.3 UNSYNCHRONIZED MUTABLE DATA SHARING

```java
   // add a value to the shared array
   public void add(int value) {
      int position = writeIndex; // store the write index

      try {
         // put thread to sleep for 0-499 milliseconds
         Thread.sleep(generator.nextInt(500));
      }
      catch (InterruptedException ex) {
         Thread.currentThread().interrupt(); // re-interrupt the thread
      }

      // put value in the appropriate element
      array[position] = value;
      System.out.printf("%s wrote %2d to element %d.%n",
         Thread.currentThread().getName(), value, position);

      ++writeIndex; // increment index of element to be written next
      System.out.printf("Next write index: %d%n", writeIndex);
   }
```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 2 of 3.)

23.4.3 UNSYNCHRONIZED MUTABLE DATA SHARING

```java

   // used for outputting the contents of the shared integer array
   @Override
   public String toString() {
      return Arrays.toString(array);
   }
}
```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 3 of 3.)

23.4.3 UNSYNCHRONIZED DATA SHARING

- Class **`ArrayWriter`** (Fig. 23.6) implements the interface **`Runnable`** to define a task for inserting values in a **`SimpleArray`** object.
- The task completes after three consecutive integers beginning with **`startValue`** are inserted in the **`SimpleArray`** object.

23.4.3 UNSYNCHRONIZED DATA SHARING

```java
// Fig. 23.6: ArrayWriter.java
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

public class ArrayWriter implements Runnable {
   private final SimpleArray sharedSimpleArray;
   private final int startValue;

   public ArrayWriter(int value, SimpleArray array) {
      startValue = value;
      sharedSimpleArray = array;
   }

   @Override
   public void run() {
      for (int i = startValue; i < startValue + 3; i++) {
         sharedSimpleArray.add(i); // add an element to the shared array
      }
   }
}
```

Fig. 23.6 | Adds integers to an array shared with other `Runnable`s. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.)

23.4.3 UNSYNCHRONIZED DATA SHARING

- Class **`SharedArrayTest`** (Fig. 23.7) executes two **`ArrayWriter`** tasks that add values to a single **`SimpleArray`** object.
- **`ExecutorService`**'s **`shutDown`** method prevents additional tasks from starting and to enable the application to terminate when the currently executing tasks complete execution.
- We'd like to output the **`SimpleArray`** object to show you the results *after* the threads complete their tasks.
  - So, we need the program to wait for the threads to complete before **`main`** outputs the **`SimpleArray`** object's contents.
  - Interface **`ExecutorService`** provides the **`awaitTermination`** method for this purpose—returns control to its caller either when all tasks executing in the **`ExecutorService`** complete or when the specified timeout elapses.

23.4.3 UNSYNCHRONIZED DATA SHARING

```java
// Fig. 23.7: SharedArrayTest.java
// Executing two Runnables to add elements to a shared SimpleArray.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest {
   public static void main(String[] arg) {
      // construct the shared object
      SimpleArray sharedSimpleArray = new SimpleArray(6);

      // create two tasks to write to the shared SimpleArray
      ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
      ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);

      // execute the tasks with an ExecutorService
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(writer1);
      executorService.execute(writer2);

      executorService.shutdown();
```

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array—the italicized text is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 1 of 3.)

23.4.3 UNSYNCHRONIZED DATA SHARING

```java

         try {
            // wait 1 minute for both writers to finish executing
            boolean tasksEnded =
               executorService.awaitTermination(1, TimeUnit.MINUTES);

            if (tasksEnded) {
               System.out.printf("%nContents of SimpleArray:%n");
               System.out.println(sharedSimpleArray); // print contents
            }
            else {
               System.out.println(
                  "Timed out while waiting for tasks to finish.");
            }
         }
         catch (InterruptedException ex) {
            ex.printStackTrace();
         }
      }
   }
```

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array—the italicized text is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 2 of 3.)

23.4.3 UNSYNCHRONIZED DATA SHARING

```
pool-1-thread-1 wrote  1 to element 0.  — pool-1-thread-1 wrote 1 to element 0
Next write index: 1
pool-1-thread-1 wrote  2 to element 1.
Next write index: 2
pool-1-thread-1 wrote  3 to element 2.
Next write index: 3
pool-1-thread-2 wrote 11 to element 0.  — pool-1-thread-2 overwrote element 0's value
Next write index: 4
pool-1-thread-2 wrote 12 to element 4.
Next write index: 5
pool-1-thread-2 wrote 13 to element 5.
Next write index: 6

Contents of SimpleArray:
[11, 2, 3, 0, 12, 13]
```

Fig. 23.7 | Executing two `Runnables` to add elements to a shared array—the italicized text is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 3 of 3.)

23.4.4 SYNCHRONIZED MUTABLE DATA SHARING – MAKING OPERATIONS ATOMIC

- The output errors of Fig. 23.7 can be attributed to the fact that the shared object, **`SimpleArray`**, is not thread safe.
- If one thread obtains the value of **`writeIndex`**, there is no guarantee that another thread cannot come along and increment **`writeIndex`** before the first thread has had a chance to place a value in the array.
- If this happens, the first thread will be writing to the array based on a stale value of **`writeIndex`**—a value that is no longer valid.

23.4.4 SYNCHRONIZED MUTABLE DATA SHARING – MAKING OPERATIONS ATOMIC

- An atomic operation cannot be divided into smaller suboperations.
- Can simulate atomicity by ensuring that only one thread carries out the three operations at a time.
- Atomicity can be achieved using the **`synchronized`** keyword.

23.4.4 SYNCHRONIZED MUTABLE DATA SHARING – MAKING OPERATIONS ATOMIC

- Figure 23.8 displays class **`SimpleArray`** with the proper synchronization.
- Identical to the **`SimpleArray`** class of Fig. 23.5, except that add is now a **`synchronized`** method—only one thread at a time can execute this method.
- We reuse classes **`ArrayWriter`** (Fig. 23.6) and **`SharedArrayTest`** (Fig. 23.7) from the previous example.
- We output messages from **`synchronized`** blocks for demonstration purposes
  - I/O *should not* be performed in **`synchronized`** blocks, because it's important to minimize the amount of time that an object is "locked."
- We call **`Thread`** method **`sleep`** to emphasize the unpredictability of thread scheduling.
  - *Never* call **`sleep`** while holding a lock in a real application.

23.4.4 SYNCHRONIZED MUTABLE DATA SHARING – MAKING OPERATIONS ATOMIC

```java
// Fig. 23.8: SimpleArray.java
// Class that manages an integer array to be shared by multiple
// threads with synchronization.
import java.security.SecureRandom;
import java.util.Arrays;

public class SimpleArray {
   private static final SecureRandom generator = new SecureRandom();
   private final int[] array; // the shared integer array
   private int writeIndex = 0; // index of next element to be written

   // construct a SimpleArray of a given size
   public SimpleArray(int size) {
      array = new int[size];
   }

```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 4.)

23.4.4 SYNCHRONIZED MUTABLE DATA SHARING – MAKING OPERATIONS ATOMIC

```java
   // add a value to the shared array
   public synchronized void add(int value) {
      int position = writeIndex; // store the write index

      try {
         // in real applications, you shouldn't sleep while holding a lock
         Thread.sleep(generator.nextInt(500)); // for demo only
      }
      catch (InterruptedException ex) {
         Thread.currentThread().interrupt();
      }

      // put value in the appropriate element
      array[position] = value;
      System.out.printf("%s wrote %2d to element %d.%n",
         Thread.currentThread().getName(), value, position);

      ++writeIndex; // increment index of element to be written next
      System.out.printf("Next write index: %d%n", writeIndex);
   }
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 4.)

23.4.4 SYNCHRONIZED MUTABLE DATA SHARING – MAKING OPERATIONS ATOMIC

```java

   // used for outputting the contents of the shared integer array
   @Override
   public synchronized String toString() {
      return Arrays.toString(array);
   }
}
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 3 of 4.)

23.4.4 SYNCHRONIZED MUTABLE DATA SHARING – MAKING OPERATIONS ATOMIC

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-2 wrote 11 to element 1.
Next write index: 2
pool-1-thread-2 wrote 12 to element 2.
Next write index: 3
pool-1-thread-2 wrote 13 to element 3.
Next write index: 4
pool-1-thread-1 wrote  2 to element 4.
Next write index: 5
pool-1-thread-1 wrote  3 to element 5.
Next write index: 6

Contents of SimpleArray:
[1, 11, 12, 13, 2, 3]
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 4 of 4.)